Quadruped Robot Locomotion with Reinforcement Learning
Neural Network Code and Report (C&R)

GROUP 18: HO KAE BOON (A0219811H), HUANG WENBO (A0285085W), ZENG ZHENGTAO (A0285134E), ZHONG HAO (A0284914W)

For the neural network component of our project, we primarily utilized the Tensorflow 1.11.0 library, which operates on a graph execution model, meaning that we need to define the complete computational graph before running forward or backward passes. This includes defining input placeholders, constructing the neural network structure, identifying trainable variables, specifying loss functions, defining optimizers, trainers, and operations for updating variables. We implemented these definitions inside the *DDPGNet* class, which can be found in the "DDPGNet.py" file. Following this, we were able to initialize a TensorFlow session and global variables, enabling us to proceed with forward and backward passes as demonstrated in the "NN_demo_code.ipynb" file. To provide a better understanding of our neural network design, we will introduce the learning algorithm our group aims to implement: Deep Deterministic Policy Gradient (DDPG), which is made up of the following components:

• Replay Buffer and Sample Batch:
DDPG is an off-policy learning algorithm, meaning the agent learns from its own past experiences, i.e. <current observation, action, next_observation, reward> at each time step. In the *DDPGNet* class, we defined an attribute called *self.replay_buffer*, which is a list to store these experiences. For each time step, we collect an experience by executing a *step()* operation in our gym environment. We preprocess this experience using the *scale_action()* and *scale_observation()* functions, which involve scaling and clipping, so that the data falls within the range of [-1, 1]. Subsequently, we append the processed experience to our replay buffer using the class method *store_to_buffer()*. Over a series of time steps, we will accumulate a collection of experiences in the replay buffer. To train our neural network, we randomly sample a batch of experiences with a size = BATCHSIZE from the replay buffer and reshape their dimensions to match the input dimension of the neural network. These operations are handled by the class method *sample_batch()*.

• Actor-Network & Critic-Network:

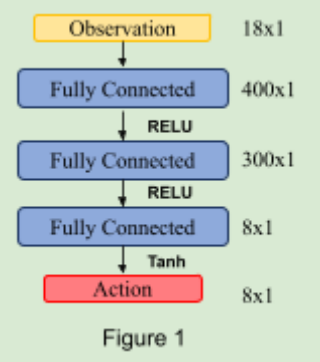

Figure 1

DDPG has the structure of an Actor-Critic network.
The Actor Network is used to determine the optimal policy. It takes in an observation as input and produces an action that maximizes the expected cumulative rewards.
Figure 1 on the left shows the architecture of our Actor Network. It comprises three fully connected (FC) layers with 400, 300, and 8 neurons, respectively. ReLU activation functions are applied to the first two FC layers, while the third FC layer utilizes the Tanh activation function to scale the output within the range of -1 to 1. These output values can be upscaled later if necessary.

The Critic network takes both the observation and the corresponding optimal action as inputs and predicts the expected future reward (Q-value). The optimal action is typically obtained from the output of the Actor Network. Our Critic network consists of four fully connected (FC) layers with 400, 300, 300, and 1 neuron(s), respectively, in addition to an "add" layer. Currently, we only utilize ReLU activation functions after the first FC layer and following the "add" operation.

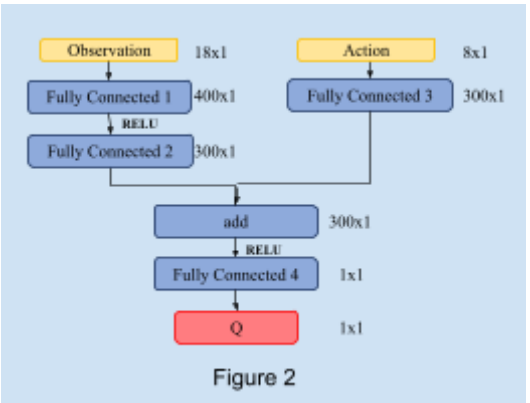

Figure 2

To implement these networks, we initialize the *DDPGNet* class with several placeholders, including observation input placeholders with dimensions (None, 18, 1) and action input placeholders with dimensions (None, 8, 1). We initialize the weights using a random normal distribution (μ=0, σ=0.0005), and the biases are set to a constant value of 0.1. Our Actor and Critic Networks are constructed by invoking the *build_actor_net()* and *build_critic_net()* methods, respectively. In both methods, we specify each fully connected layer by providing the input placeholder, output dimension, initialized weights, biases, activation functions, and a unique scope (either "actor" or "critic"). The scope is important as it is used to distinguish between the trainable variables associated with different networks."

- Networks Training, Target Networks:

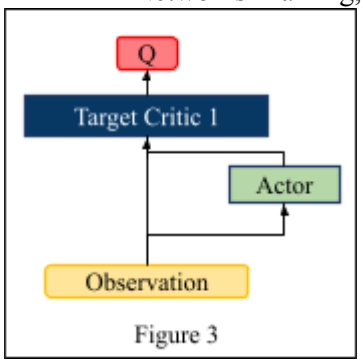

Figure 3

With reference to Figure 3 on the left, the training process of the Actor is as follows: Starting with the current observation, the Actor produces an action. Subsequently, we supply this action, along with the observation, to the Critic to estimate the Q value. The Actor Loss is then computed as - Q. This loss serves as the signal for the backward pass, with the objective of minimizing the Actor Loss accomplished through the use of the Adam Optimizer.

To mitigate fluctuations and facilitate convergence, we replace the Critic with a Target Critic during the backward pass. The Target Critic's variables are temporarily frozen, and only the Actor's variables are updated.

With reference to Figure 4 on the right, the training process of Critic is as follows: Given the current observation and action, the Critic evaluates the Q value. On the other hand, we provide the next_observation to the Actor to obtain the next ideal action. We then feed this action, along with the next observation, to the Critic to compute the Q_next value. The Target_Q value can be computed as (Reward + GAMMA * Q_next). With this value, we can compute the Critic Loss, which is the mean square error between the Q value and the target_Q. Finally, a backward pass is performed to minimize the Critic Loss using Adam Optimizer.

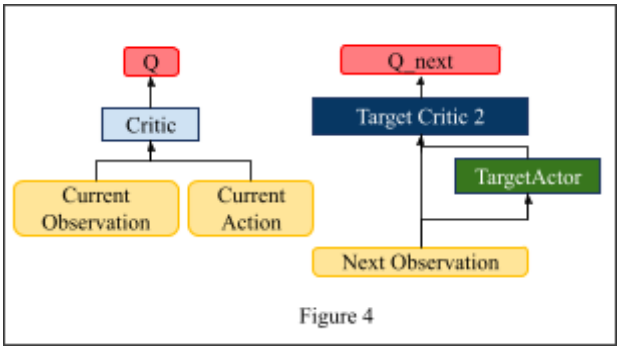

Figure 4

During the backward pass, we replace both the Actor and Critic that are involved in the calculation of Q_next with their "Target" versions. All the Target networks are temporarily frozen, and only Critic's variables are updated. Do note that in our project, we employ two distinct Target Critics: Target Critic 1 is used during the training of the Actor, whereas Target Critic 2 is utilized during the training of the Critic.

Following each backward pass of the Actor and Critic, we execute a soft update on the Target Networks. This update is performed using the equation: Target network's new weight = TAU * Main network's weight + (1 - TAU) * Target network's old weight. TAU is the hyperparameter that controls the update rate.

To implement the target networks, we introduced additional placeholders during the initialization of our *DDPGNet* class. These placeholders include the next_observation placeholder with dimensions (None, 18, 1) and the reward placeholder with dimensions (None, 1, 1). Subsequently, we constructed the Target Actor, Target Critic 1, and Target Critic 2 by invoking the *build_target_actor_net()*, *build_target_critic_1_net()*, and *build_target_critic_2_net()* methods, respectively. In each of these methods, we specify each fully connected layer by providing their input placeholder, output dimension, initialized weights, biases, activation functions, and a unique scope (either "target_actor," "target_critic_1," or "target_critic_2").

As for backward passes, we obtain the trainable variables of different networks by specifying the scope limit. Then, we define the actor loss, actor optimizer, actor trainer, critic loss, critic optimizer, and critic trainer accordingly. When defining the losses, we provide their respective learning rates (CRITIC_LR or ACTOR_LR). When defining the trainer, we provide the trainable variable list to ensure that only the main network's variables are updated during training. For the soft update process, we iterate through each target variable and define the update operation. All of these update operations are stored in the *update_target_ops* list.

Reflection:

Throughout this project, our team has acquired a deeper understanding of neural networks and their implementation using TensorFlow. For this project, we have made an adjustment to our OpenAI code, by changing our action from discrete to continuous. We introduced a new method *get_random_action()* that generates random continuous action values within the range of [-26.5, 26.5]. These values correspond to the real-life aliengo motor torque range. Our current exploration is done via random action. In the future, we plan to study the Ornstein-Uhlenbeck Process to introduce correlated actor noise and parameter noise to the Actor network, promoting more exploration. We also found it necessary to fine-tune our reward mechanism to enhance the training process.